

C0 Reference

15-122: Principles of Imperative Computation
Frank Pfenning

Monday 18th January, 2021
Compiler revision 707
([updates](#) since January 30, 2011)

Contents

1	Introduction	3
2	Types	3
3	Integers	4
4	Booleans	5
5	Functions	6
6	Commands	7
6.1	Assignments	7
6.2	Expressions as Statements	7
6.3	Conditionals	7
6.4	Loops	8
6.5	Blocks	8
6.6	Returns	9
6.7	Assertions	9
6.8	Errors	9
7	Characters	9
8	Strings	10
9	Arrays	10

Reference	C0.2
10 Pointers	11
11 Structs	11
12 Compiler Directives	13
13 Contracts	13
13.1 Function Contracts	14
13.2 Loop Invariants	15
13.3 Assertions	15
13.4 Purity	15
14 Grammar	16
14.1 Lexical tokens	16
14.2 Grammar	18
14.3 Annotations	18
15 C1 Language Extension	18
15.1 Advanced Control Constructs	18
15.2 Generic Pointers	21
15.3 Function Pointers	22
16 Static Semantics Reference	24
17 Updates	26

1 Introduction

The programming language C0 is a carefully crafted subset of the C aimed at teaching introductory algorithms and imperative programming. It is restricted enough to permit a straightforward safe implementation in which attempts to access an array element out of bounds array can be reliably detected. This eliminates a class of insidious bugs that are difficult to diagnose or detect through testing, as numerous security problems with commercial software attest. As a consequence the language soundly supports a conservative garbage collector to reclaim memory, rather than relying on error-prone explicit memory management. It is intended that all operations are completed defined, although even correct programs may abort when they run out of memory. The combination of these features allow us to soundly reason about contracts and loop invariants, which constitute a new language layer. More about the differences and the transition to C in a separate document; in this document we introduce the language itself. We assume the reader has some familiarity with programming, but not necessarily with C. This document is not intended as a tutorial, but as a concise reference.

2 Types

Programming languages can be characterized by the types that they support, and C0 is no exception. We briefly list the types and explain their meaning before discussing the constructions associated with each form of type in turn.

int The only numeric type supported in C0 is `int`. Values of this type are 32-bit words, interpreted as integers according to two's complement representation. Computations are performed modulo 2^{32} .

bool The type of booleans has just two elements, `true` and `false`. They are used for conditions, as well as contracts.

char The type of characters contains ASCII (not Unicode) characters written as `'c'`.

string The type of strings contains constant strings of characters, written as `"this is a string"`.

$t []$ For any type t , we can form the type $t []$, the type of arrays of values of type t . A value of this type is a reference to an array stored in memory. An array A has an intrinsic length n determined at the type of allocation; its elements can be referenced as $A[0]$ through $A[n - 1]$.

$t *$ For any type t , we can form the type $t *$, the type of pointers to values of type t stored in memory. Its values are addresses and a special value NULL.

struct s Structs are aggregates whose members can be accessed through field names. The fields and their types for each structure named s have to be explicitly declared. Structures are allocated in memory. Unlike the elements of other types, structures cannot be assigned to variables or passed as function arguments because they can have arbitrary size. Instead, we pass pointers to structs or arrays of structs.

Functions and commands There are no explicit types for functions and commands, because the language does not allow them to be passed as arguments or stored explicitly. Of course, the language has means to define and invoke functions, and execute commands including variable assignments, conditionals and loops.

Contracts Again, there is no explicit type for contracts, but C0 supports contracts governing the permissible invocations, return values, and effects of functions. Currently, they can only be checked dynamically, although some tools for static checking are under development.

3 Integers

The type of integers is `int`. The values are 32-bit words, interpreted according to two's complement arithmetic. This means arithmetic is modulo 2^{32} , with the minimal representable integer being $-2^{31} = -2147483648$ and the maximal being $2^{31} - 1 = 2147483647$. Decimal constants c in a program must be in the range $0 \leq c \leq 2^{31}$, where $2^{31} = -2^{31}$ according to modular arithmetic. Hexadecimal constants must fit into 32 bits.

Integer operations are the usual binary $+$ (addition), $-$ (subtraction), $*$ (multiplication), which operate modulo 2^{32} . In addition we have integer division n/k and modulus $n\%k$. Division truncates towards zero, and both division and modulus raise an overflow exception if $k = 0$ or $n = -2^{31}$ and $k = -1$. If n is negative, the result of the modulus will be negative, so that $(n/k)*k + n\%k == n$ when the left-hand side doesn't overflow.

Comparisons `<`, `<=`, `>=`, `>` return a boolean when applied to integers, as do `==` (equality) and `!=` (disequality).

We can also view and manipulate values of type `int` as 32-bit words. For this purpose, we have a hexadecimal input format. A number constant in hexadecimal form starts with `0x` and contains digits `0` through `9` and `a` through `f`. Hexadecimal digits are not case sensitive, so we can also use `X` and `A` through `F`.

Binary bitwise operations on values of type `int` are `&` (and), `^` (exclusive or), `|` (or), and we also have unary bitwise complement `~`. The hybrid shift operators `n << k` and `n >> k` shift the bits of `n` by `k`. Here, `k` is required to be in the range from 0 to 31, inclusively. Otherwise, it raises an arithmetic exception. On the left shift, the lower bits are filled with 0; on the right shift the higher bit is copied. This means that left shift by `k` is equal to multiplication by 2^k , and right shift `k` is like division by 2^k , except that it truncates towards $-\infty$ rather than 0.

The default value for integers, which is needed for some allocation operations, is 0.

The precedence and associativity of the operators is shown in Figure 3. In general, expressions are guaranteed to be evaluated from left-to-right so that, for example, in `f(x) + g(x)` we first call `f` and then `g`. Any effects such as input/output are guaranteed to happen in the specified order.

4 Booleans

The type `bool` is inhabited by the two values `true` and `false`.

Booleans can be combined with logical (as opposed to bit-wise) conjunction `&&` (and) and disjunction `||` (or), which are binary operators. Their evaluation short-circuits in the sense that in `b && c`, if `b` evaluates to `false`, then `c` is not evaluated. Similarly, in `b || c`, if `b` evaluates to `true`, then `c` is not evaluated. There is also a unary operator of logical negation `!` (not).

Booleans can be tested with a conditional expression (also called *ternary operator*) `b ? e1 : e2` which first evaluates `b`. If `b` is `true`, it then evaluates `e1` and returns its value, otherwise it evaluates `e2` and returns its value. In conditional expressions, both branches `e1` and `e2` must have the same type, and this type must be *small* (see the sections on functions and structs).

Booleans can be compared for equality (`==`) and disequality (`!=`).

The default value is `false`.

5 Functions

Functions are not first-class in C0, but can only be declared or defined at the top-level of a file. A function definition has the form

```
t g (t1 x1, ..., tn xn) { body }
```

where t is the result type of the function called g which takes n arguments of type t_1, \dots, t_n . The scope of parameters x_1, \dots, x_n is *body*, which is a block consisting of a sequence of additional local variable declarations followed by a sequence of statements. Note that function definitions are *not* terminated by a semi-colon. The scope of the function name g include *body* and the remainder of the compilation unit, typically a file. Currently, if multiple files are given to the compiler they are concatenated sequentially into a single compilation unit.

Argument and result types must be *small*, which means that they cannot be structs. Instead of structs, programs should pass either pointers to structs or arrays containing structs.

Functions may be declared without giving a definition in the form

```
t g (t1 x1, ..., tn xn);
```

which allows the use of g in subsequent functions in the same compilation unit.

A function may be declared multiple times in a compilation unit. Those that are referenced (and not in a library) must be defined exactly once. Multiple declarations must be consistent, and consistent with a possible definition, but can differ in the name of the parameters and contracts.

Library functions are special in that they may be declared in a library header file `<lib>.h0` for library `<lib>`, but they cannot be defined. Libraries can be included on the command line using the switch `-l<lib>`. See a separate description of the compiler interface.

Expressions denoting function calls have the form $g(e_1, \dots, e_n)$. The arguments e_1, \dots, e_n are evaluated in sequence from left to right and the resulting values passed to the function g .

Functions may recursively call themselves. As a language, C0 does not specify a minimum or maximum required recursion depth. Different implementations may impose different limits, or no limit at all.

6 Commands

Commands are not first-class in C0, but occur in the bodies of functions. We have *assignments*, *conditionals*, *loops*, *blocks*, and *returns*.

6.1 Assignments

Basic assignments $x = e$; assign to x the value of the expression e . The types of x and e must match for the assignment to be legal.

More generally, the left-hand side of an assignment can be an *lvalue* which includes additional ways of referencing memory. Besides variables, the other possible lvalues are explained below for arrays ($lv[e]$), pointers ($*lv$), and structs ($lv.f$). In assignment $lv = e$, the left-hand side lv is evaluated first, then e , and then the assignment is attempted (which may fail based on the form of lv , for arrays or pointers).

There are also compound assignments of the form $lv\ op = e$ which translate to $lv = lv\ op\ e$ where op is a binary operator among $+$, $-$, $*$, $/$, $\%$, $\&$, \wedge , $|$, \ll , or \gg , except that lv is evaluated only once.

Finally, there compound assignments $lv++$ and $lv--$ which desugar into $lv += 1$ and $lv -= 1$, respectively.

6.2 Expressions as Statements

An expression e ; can be used as a statement. Such a statement evaluates e , incurring all of its effects, and then discards the return value if there is any.

6.3 Conditionals

Conditionals have the form **if** (e) s_1 **else** s_2 . Note that conditionals (like loops) are *not* terminated by a semi-colon. The condition e must be of type `bool`. It is evaluated first, followed either by s_1 (if e is true) or s_2 (if e is false).

There is a shorthand, **if** (e) s_1 , which omits the **else**-clause, which translates into **if** (e) s_1 **else** $\{\}$ where $\{\}$ is the empty block which has no effect. The possibility to omit **else**-clauses creates an ambiguity because with two **ifs** and only one **else** it may be unclear which condition the **else** is associated with. For example,

```
if (e1) if (e2) s1 else s2
```

could be read as

```
if (e1) {if (e2) s1} else s2
```

or

```
if (e1) {if (e2) s1 else s2}
```

The rule is that an **else**-clause is matched up with the most recent **if** that does not have an **else**-clause while respecting explicit grouping into blocks, so the second reading is the correct one.

6.4 Loops

There are two forms of loops.

```
while (e) s
```

begins by evaluating e . If e evaluates to `true` it continues with the execution of s , subsequently testing e again. If e evaluates to `false` we finish the **while** loop and continue with the next statement.

```
for (s1; e; s2) s3
```

begins by evaluating the loop initializer s_1 which must be a simple statement, usually an assignment or a variable declaration. Then it evaluates e . If e is `true`, we execute the body s_3 followed by the step expression s_2 , which must again be a simple statement (but may not be a variable declaration), followed in turn by the exit test e . Both s_1 and s_3 may be omitted, in which case they act like the empty command which immediately finishes without an effect. If s_1 is a variable declaration, the scope of the declared variable consists of e , s_2 , and s_3 .

The two control commands that can affect the execution of a loop, **break** and **continue**, are only available in the extended language standard C1, but are reserved keywords in C0. See Section 15.

6.5 Blocks

Blocks have the form $\{ss\}$, where ss is a (possibly empty) sequence of statements. The statements in ss are executed in order.

One particular form of statement is a variable declaration, which has one of the two forms

```
t x;
```

where t is a type and x is a variable, or

```
t x = e;
```


where t is a type, x is a variable, and e is an expression initializing x which must have type t .

In either form, the scope of x consists of the remaining declarations and statements in the block containing the declaration.

Variables declared in an outer scope (either as function parameters of an enclosing block) can not be declared again in an inner block with the same name.

6.6 Returns

Anywhere in the body of a function there can be a **return** statement, either of the form **return** e ; for an expression e or just **return**;

In the form **return** e ;, the type of e must match the result type of the function. In the form **return**;;, the result type of the function must be the special type `void` that can only be used to indicate that a function does not return a value. Such functions can only be called as expressions that appear as statements.

6.7 Assertions

An assertion statement has the form

assert(e);

where e is a boolean test. If e evaluates to `false`, an error message is issued and the computation is aborted. Assertion statements are *always* executed, no matter whether contracts are checked dynamically (see Section 13).

6.8 Errors

An error statement has the form

error(s);

where s is a string expression. Errors are intended to be used as a response to bad user input, whereas failed assertions indicate an internal error of the program. Executing an error statement will print the string and then immediately terminate the program.

7 Characters

Characters are a special type to represent components of strings. They are written in the form `'c'`, where c can be any printable ASCII character, as

well as the following escape sequences `\t` (tab), `\r` (return), `\f` (formfeed), `\a` (alert), `\b` (backspace), `\n` (newline), `\v` (vertical tab), `\'` (quote), `\"` (doublequote), `\0` (null). The default value for characters is `\0`. Characters can be compared with `==`, `!=`, `<`, `<=`, `>=`, `>` according to their ASCII value, which is always in the range from 0 to 127, inclusively.

8 Strings

Strings have the form `"c1...cn"`, where c_1, \dots, c_n is an ASCII character as above, including the legal escape sequences except for null (`\0`), which may not appear in strings. The double-quote character itself `"` must be quoted as `\"` so it is not interpreted as the end of the string. The default value for type **string** is the empty string `""`. Strings can not be compared directly with comparison operators, because in a language such as C the comparison would actually apply to the addresses of the strings in memory, with unpredictable results. Appropriate comparison functions are provided by the string library.

9 Arrays

The type of arrays with elements of type t is denoted by `t []`. Arrays must be explicitly allocated in memory, but they do not need to be deallocated, a function performed by the garbage collector. For this purpose we have a new expression

alloc_array(t , e)

which returns a reference to a new array of type `t []`. The expression e must evaluate to a non-negative integer n denoting the length of the allocated array. Elements of an array A allocated in this way are accessed as `A[0]` through `A[n-1]`. Attempt to index an array with a negative number or a number greater or equal to n will result in an array bounds violation that will terminate the program.

Array references can also be used as lvalues. For example, `A[0] = 1` will write 1 to the first element of A which must be an integer array, and `A[2]++` will increment the third element in the array.

For every type t there is a distinguished zero-sized array of type `t []` which serves as the default. Because its size is zero, the only operations that can be performed on this element are comparisons for equality (`==`) and disequality (`!=`).

It is important to remember that comparisons of variables of type `t []` are comparisons of array references, and not the array elements, and similarly for argument passing and variable assignment.

Because of its roots in C, one cannot determine the length of arrays in programs. This allows an unsafe implementation in which array bounds of accesses are not checked, a low-level efficiency improvement that can make a significant difference in certain kinds of highly numerical code. On the other hand, contracts must be able to mention the length of arrays to ensure the absence of runtime errors. For that purpose there is a special function `\length(e)` that can only be used in contracts. When contracts are to be checked dynamically, the compiler will take care to represent arrays such that the length is stored.

10 Pointers

The type of pointers of type t is denoted by t^* . We obtain a pointer to a memory location holding a value of type t using the new expression

alloc(t)

We dereference pointers using the expression $*e$ which evaluates to a value of type t if e evaluates to a pointer of type t^* .

Pointers also introduce a new lvalue $*lv$ which references the memory location or variable denoted by lv .

For each type t there is a special pointer `NULL` of type t^* . Attempts to dereference `NULL` will result in a runtime exception that terminates the program. `NULL` is the default value for pointers at each type.

The constant `NULL` introduces a type ambiguity which can be locally resolved in expressions where relevant. For example, a function call `f(NULL)` is well-typed if `f` expects an argument of type `int*` or `bool*` or `int[]*`, etc. The one exception to this rule is code of the form `*NULL` which could be used at an arbitrary type t . In order to avoid this typical ambiguity, it is an error to write `*NULL` in programs.

11 Structs

Structs are the only types that can aggregate values of different type. We write **struct** s for the type of structs named s . Structure names occupy their own name space, as do the names of the fields of structs; neither can

conflict with names of variables, functions, or other fields or struct names. Structs with name s are defined with

```
struct s { t1 f1; ... tn fn; };
```

and have fields named f_1, \dots, f_n of types t_1, \dots, t_n , respectively. After such a declaration, the field f_i of a struct denoted by e of type **struct** s can be accessed with the expression $e.f_i$ and has type t_i .

Structs must be allocated in memory. Because they may be of large size, their value can not be held in a variable or passed as an argument to a function, or returned from a function. We therefore call **struct** s a *large* type, while all other types in the language are *small*. In particular, array types $t[]$ are small because a value of this type is a reference to an array, not the array itself. In contrast, a value of type **struct** s is the struct itself. This means that programs mostly manipulate either pointers to structs **struct** s^* or arrays of structs **struct** $s[]$. As a result there is no special form to allocate a struct: structs will be allocated as the result of allocating memory with **alloc(struct s)** or **alloc_array(struct s, e)** or other data types with embedded structs. Each of the fields of a struct allocated in this way is initialized with default values according to their type.

Because pointers to structs are common, there are two constructs supporting the idiomatic use of pointers to structs. The first is the expression $e \rightarrow f$ which stands for $(*e).f$.

The second is a general form of type definition written as

```
typedef t a
```

where t is a type and a is a type name. This definition can appear only at the top-level and the scope of the type name a is the rest of the compilation unit. In order avoid certain ambiguities in the grammar, type names a occupy the same name space as variables and functions. It is a conflict to declare or define a function or a variable with the same name as a type.

The idiomatic, but not the only use of the above, has the form

```
typedef struct s* s
```

after which the type name s represents pointers to **struct** s .

Struct types **struct** s can be used before they are defined, but they can also be explicitly declared as

```
struct s;
```

Since the fields of such a struct are not known, they cannot be accessed by a program. Nonetheless, pointers to elements of such type have a uniform

size and can therefore be passed as arguments even without knowing the precise representation of s . This allows a very weak form of polymorphism in C0.

12 Compiler Directives

As described elsewhere, the cc0 compiler for C0 processes a list of files in sequence, and can also include specific libraries as documented in the [C0 Library Reference](#). One can also include references to specific libraries and other source files in C0 source files directly. For libraries this is good style, because it makes dependencies on libraries explicit. The directive¹

```
#use <lib>
```

will load the library called *lib* before processing the remainder of the file. To load a library, the compiler will search for and process files `lib.h0` (for external libraries) and `lib.h0` and `lib.c0` (for libraries written in C0) in a set of implementation-dependent directories. The second form of the directive

```
#use "filename"
```

will load the file *filename* (typically with a `.c0` extension) and process it before processing the rest of the file.

Either form of the directive will not perform any action if the library or file has already been loaded during the current compilation process. `#use` directives must precede all other declarations in a file.

13 Contracts

Contracts collectively refer to assertions made about the code. Contracts are never necessary to execute the code, but it is possible to check the adherence to contracts dynamically by compiling the code with a special flag. Contracts specify either pre- and post-conditions for functions, loop invariants, or preconditions for statements.

From the syntactical point of view, contracts appear as special comments or *annotations* that can be ignored by a compiler that does not support them. As such, they constitute a separate language layer which is entirely absent from C. Annotations start with `//@` and extend to the end of the line, or delimited by `/*@` and `@*/`. For illustration purposes below we use the single-line form.

¹with explicit angle brackets `<` and `>`

Contracts should never have store effects and should terminate, although the compiler currently does not check that. It is permissible for contracts to raise exceptions, including the exception that the contract was not satisfied.

13.1 Function Contracts

For functions to work correctly, they often impose conditions on their input. For example, an integer square root may require its argument to be non-negative, or a dictionary insertion function may require the dictionary to be sorted. Conversely, it will make some guarantees on its outputs. For example, the integer square root should really return the root and perhaps more specifically the positive one, and the insertion function should really insert the new word into the dictionary into its proper place. The former is called a *precondition* for the function, specified with `@requires`; the latter is a *postcondition* specified with `@ensures`.

A function definition then has the general form

```
t g (t1 x1, ..., tn xn)
contracts
{ body }
```

where a contract is one of the following

```
//@requires e;
//@ensures e;
```

The expression e , which must have type `bool` can mention the parameters x_1, \dots, x_n . It can also mention the special function `\length(e)` mentioned above, the special variable `\result` in `@ensures` clauses. The body of a function may not assign to any variable that occurs in an `@ensures` clause. This means that the function contract can be correctly interpreted without knowing the body of the function.

Contracts must be in single-line or multi-line comments, as in the following example.²

```
int exp (int k, int n)
//@requires n >= 0;
//@ensures \result >= 1;
/*@ensures \result > n; @*/
{ int res = 1; int i = 0;
```

²For modular arithmetic as specified for C0, this contract is *not* satisfied because the result may be negative.

```
while (i < n) {
    res = res * k;
    i = i + 1;
}
return res;
}
```

When dynamic checking of contracts is enabled, `@requires e`; specifications are checked just before the function body and `@ensures e`; are checked just before the return, with the special variable `\result` bound to the return value.

13.2 Loop Invariants

Loop invariant annotations have the form

```
//@loop_invariant e;
```

where e has type `bool`. The general form of **while** and **for** loops is

```
while (e) invs s
for (s1; e; s2) invs s
```

where `invs` is a possibly empty sequence of invariants. As for function contracts, they must be stylized single-line or delimited comments.

When dynamic checking is enabled, the loop invariant is checked on every iteration just before the exit condition e is evaluated and tested.

13.3 Assertions

Assertion annotations have the form

```
//@assert e;
```

An assertion annotation must precede another statement and can be seen as guard on that statement. When a function is called correctly, according to its precondition (`//@requires`), the assert annotations should not fail; in that sense they express expected internal invariants of functions, just like loop invariants.

13.4 Purity

Functions that are called from annotations must be *pure*, which means that they may not modify previously allocated memory. Allowing such con-

tracts could lead to different results, depending on whether contract checking is enabled or not. Other kinds of effects, like input, output, or exceptions are permitted. The C0 language implementation performs a light analysis pass over the code to uncover functions used in annotations with side effects and gives an appropriate error message. Since purity for contract functions is at present not strictly part of the language definition, purity checking can be disabled by passing the flag `--no-purity-check`.

14 Grammar

We now summarize the grammar rules of the language.

14.1 Lexical tokens

We have the following classes of tokens: identifiers, numerical constants, string literals, character literals, separators, operators, and reserved keywords. In addition there is whitespace, which is a regular space, horizontal and vertical tab, newline, formfeed and comments. Whitespace separates tokens, but is otherwise ignored. Other control (non-printing) characters in the input constitute an error.

Comments may be on a single line, starting with `//` and ending with newline, or delimited, starting with `/*` and ending with `*/`. Delimited comments must be properly nested. When annotations are parsed and checked, the first character of a comment must not be `@`, which would start an annotation.

Compiler directives are always on a single line and have the form `#use` followed by whitespace and then either a library literal `<liblit>` or a string literal `<strlit>`. Other top-level directives starting with `#` are ignored, but may produce a warning.

We present the token classes as regular expressions. [Square brackets] surround enumerations of single characters or character ranges like `a-z`, `<angle brackets>` surround nonterminals in the grammar.

The reserved keywords of the language are:

```
int bool string char void struct typedef  
if else while for continue break return assert  
error true false NULL alloc alloc_array
```



```

<id>          ::= [A-Za-z_][A-Za-z0-9_]*

<num>         ::= <decnum> | <hexnum>
<decnum>      ::= 0 | [1-9][0-9]*
<hexnum>      ::= 0[xX][0-9a-fA-F]+

<strlit>      ::= "<schar>*"
<chrlit>      ::= '<cchar>'
<liblit>      ::= <<lchar>*>
<schar>       ::= <nchar> | <esc>
<cchar>       ::= <nchar> | <esc> | " | \0
<nchar>       ::= (normal printing character except ")
<lchar>       ::= (normal printing character except >)
<esc>         ::= \n | \t | \v | \b | \r | \f | \a
               | \\ | \' | \"

<sep>         ::= ( | ) | [ | ] | { | } | , | ;
<unop>        ::= ! | ~ | - | *
<binop>       ::= . | -> | * | / | % | + | - | << | >>
               | < | <= | >= | > | == | !=
               | & | ^ | | | && | || | ? | :
<asnop>       ::= = | += | -= | *= | /= | %= | <<= | >>=
               | &= | ^= | |=
<postop>      ::= -- | ++

```

Figure 1: C0 lexical tokens

14.2 Grammar

We present the grammar in a similarly abbreviated style in Figure 2. Here, [brackets] surround optional constituents. Identifiers occupy four name spaces: variables and function names <vid>, type names <aid>, struct names <sid>, field names <fid>. Variable and function names may not conflict with type names; otherwise the same identifiers can be reused.

14.3 Annotations

Annotations may be on a single line, starting with `//@` and ending with newline, or delimited, starting with `/*@` and ending with `@*/`. In an annotation, the `@` character is treated as whitespace.

The additional reserved keywords are

requires ensures loop_invariant \result \length \hastag

The grammar is modified by adding the following cases. The restrictions on annotations are detailed in Section 13.

This extension introduces another ambiguity, because a statement of the form `<anno> <anno> <stmt>` could be one statement with two annotations, or an annotated annotated statement. We resolve this by always interpreting it as a single statement with two annotations, or multiple annotations in the general case.

15 C1 Language Extension

The C1 language extension of C0 contains some experimental features, which are automatically permitted when the compiler is invoked on files with the `.c1` file extension.

15.1 Advanced Control Constructs

As of September 1, 2013 (svn revision 350), the C1 language standard supports the **break** and **continue** control constructs, previously available in C0.

```
<stmt> ::= ... | break ; | continue ;
```

Execution of the **break** statement immediately exits the innermost loop that contains it and proceeds with the statement following the loop. The loop invariant is not checked.

```

<prog> ::= (<gdecl> | <gdefn>)*

<gdecl> ::= struct <sid> ;
           | <tp> <vid> ( [<tp> <vid> (, <tp> <vid>)*] ) ;
           | #use <liblit> \n | #use <strlit> \n

<gdefn> ::= struct <sid> { (<tp> <fid> ;)* } ;
           | <tp> <vid> ( [<tp> <vid> (, <tp> <vid>)*] ) { <stmt>* }
           | typedef <tp> <aid> ;

<stmt> ::= <simple> ;
           | if ( <exp> ) <stmt> [ else <stmt> ]
           | while ( <exp> ) <stmt>
           | for ( [<simple>] ; <exp> ; [<simple>] ) <stmt>
           | return [<exp>] ;
           | { <stmt>* }
           | assert ( <exp> ) ;
           | error ( <exp> ) ;

<simple> ::= <lv> <asnop> <exp>
           | <lv> ++
           | <lv> --
           | <exp>
           | <tp> <vid> [= <exp>]

<lv> ::= <vid> | <lv> . <fid> | <lv> -> <fid>
        | * <lv> | <lv> [ <exp> ] | ( <lv> )

<tp> ::= int | bool | string | char | void
        | <tp> * | <tp> [ ] | struct <sid> | <aid>

<exp> ::= ( <exp> )
        | <num> | <strlit> | <chrlit> | true | false | NULL
        | <vid> | <exp> <binop> <exp> | <unop> <exp>
        | <exp> ? <exp> : <exp>
        | <vid> ( [<exp> (, <exp>)*] )
        | <exp> . <fid> | <exp> -> <fid>
        | <exp> [ <exp> ]
        | alloc ( <tp> ) | alloc_array ( <tp> , <exp> )

```

Figure 2: C0 Grammar, without annotations

Operator	Associates	Meaning
() [] -> .	left	parens, array subscript, field dereference, field select
! ~ - * ++ --	right	logical not, bitwise not, unary minus, pointer dereference increment, decrement
* / %	left	integer times, divide, modulo
+ -	left	plus, minus
<< >>	left	(arithmetic) shift left, right
< <= >= >	left	comparison
== !=	left	equality, disequality
&	left	bitwise and
^	left	bitwise exclusive or
	left	bitwise or
&&	left	logical and
	left	logical or
? :	right	conditional expression
= += -= *= /= %= &= ^= = <<= >>=	right	assignment operators

Figure 3: Operator precedence, from highest to lowest

```

<spec> ::= requires <exp> ;
        | ensures <exp> ;
        | loop_invariant <exp> ;
        | assert <exp> ;

<anno> ::= //@ <spec>* \n
        | /*@ <spec>* @*/

<gdecl> ::= ...
          | <tp> <vid> ( [<tp> <vid> ( , <tp> <vid>)*] ) <anno>* ;

<gdefn> ::= ...
          | <tp> <vid> ( [<tp> <vid> ( , <tp> <vid>)*] ) <anno>*
            { <stmt>* <anno>* }

<stmt> ::= ... | <anno>+ <stmt> | { <stmt>* <anno>+ }

<exp> ::= ... | \result | \length ( <exp> )

```

Figure 4: C0 grammar extensions for annotations

Execution of the **continue** statement immediately short-circuits the body of the innermost loop that contains it and proceeds with the exit test (in a **while** loop) or the step command followed by the exit test (in a **for** loop). If contract checking is enabled, the loop invariant is checked prior to the exit test.

15.2 Generic Pointers

As of August 21, 2014 (svn revision 384), the C1 language standard supports generic pointers of type **void***. The syntax is extended by allowing a cast prefix operator of the same precedence as the other prefix operators in the grammar (see Figure 3). For use in contracts, we have the additional construct `\hastag(<tp>, <exp>)`.

```

<exp> ::= ... | (<tp>) <exp> | \hastag ( <tp> , <exp> )
<lv>  ::= ... | * (<tp>) <lv>

```

The form **(void*)***e* casts the expression *e* of type *t** to be of type **(void*)**. Operationally, this new pointer references a pair consisting of a runtime representation of the type *t** (the *tag*) and the pointer value of *e*.

The second form **(t*)***e* where *t* ≠ **void** casts an expression *e* of type

void* to have type t^* . If the tag agrees with the type t^* , it strips off the tag and returns the underlying pointer of type t^* . If the tags do not agree, an appropriate runtime exception is raised and the program is terminated.

In contracts, we can verify the value of a tag with the boolean expression $\backslash\text{has tag}(t^*, e)$ which is true if e (which must be of type **void***) has tag t^* and false otherwise. The tag can never be **void***, so t cannot be void

From Fall 2020, casts are now allowed in lvalues as well. The type cannot be void.

Casting does not affect the null pointer, which remains NULL and serves as the default value of type **void***. Therefore, we consider $\backslash\text{has tag}(t^*, \text{NULL})$ to be true for any permissible type t different from void.

15.3 Function Pointers

As of August 21, 2014 (svn revision 384), the C1 language standard also supports a limited form of function pointer. We add a new unary prefix operator $\&$ pronounced “*address of*”, which can only be applied to functions and has the same precedence as other unary prefix operators such as $*$. We can dereference a function pointer and apply it to a sequence of arguments with a new form of function call.

```
<unop> ::= ... | &
<exp> ::= ... | (* <exp>) ( [<exp> (, <exp>)*] )
```

In order to use function pointers we need to be able to assign them types. For this purpose, we allow a particular idiomatic use of **typedef** which is consistent with but much more restrictive than C and declares a *function type name* $\langle\text{fnid}\rangle$ which occupies the same name space as (ordinary) type names.

```
<gdefn> ::= ...
          | typedef <tp> <fnid> ( [<tp> <vid> (, <tp> <vid>)*] ) <anno>* ;
<tp> ::= ... | <fnid>
```

Note that this is exactly the same form as a function declaration (also called a *function prototype*) preceded by the **typedef** keyword. In particular, it also allows type annotations which can be used to impose contracts on functions of type $\langle\text{fnid}\rangle$.

Function types, named by a $\langle\text{fnid}\rangle$ are large types and, moreover, function values cannot be allocated on the stack or heap. That is, we store and pass only pointers to functions, not functions themselves. Function type names are treated *nominally*, which means that two distinct function type

names are considered different, even if their definitions happen to be the same.

Here is a small example of the use of function pointers and generic pointers to specify sortedness of an array segment of non-null generic data.

```
/** generic section */
/* define function type 'cmp' */
typedef bool cmp(void* p, void* q)
/*@requires p != NULL && q != NULL;
;

/* 'pred' is a pointer to a comparison function */
bool is_sorted(cmp* pred, void*[] A, int lower, int upper)
/*@requires 0 <= lower && lower <= upper && upper <= \length(A);
/*@requires pred != NULL;
{
    for (int i = lower; i < upper-1; i++)
        /*@loop_invariant lower <= i;
        if (!(*pred)(A[i], A[i+1])) /* call function 'pred' */
            return false;
    return true;
}

/** specific instance */
bool leq(void* p, void* q)
/*@requires p != NULL && q != NULL;
/*@requires \hastag(int*, p) && \hastag(int*, q);
{
    return *(int*)p <= *(int*)q;
}

int main() {
    int n = 10;
    void*[] A = alloc_array(void*, n);
    for (int i = 0; i < n; i++) {
        int* p = alloc(int); *p = i;
        A[i] = (void*)p;
    }
    return is_sorted(&leq, A, 0, n) ? 1 : 0;
}
```

16 Static Semantics Reference

The static semantics enforces the following conditions.

- `#use` directives must precede all other declarations.
- All operators and functions are used with the correct number of arguments of the correct type, as explained in the sections on the various language constructs.
- Operators `<`, `<=`, `>=`, and `>` are overloaded in that they apply to type **int** and **char**. Both sides must have the same type.
- Operators `==` and `!=` are overloaded in that they apply to types **int**, **bool**, **char**, `t []`, and `t *`. They do not apply to arguments of type **string** and **struct** s. Both sides must have the same type.
- Structs cannot be passed to or from functions or assigned to variables.
- All control-flow paths in the body of each function end with a return statement of the correct type, unless the function has result type **void**.
- Every variable must be declared with its type.
- Along each control-flow path in the body of each block in each function, each locally declared variable is initialized before its use.
- Function parameters and locally declared variables with overlapping scopes may not have the same name.
- Names of functions or variables may not collide with the names of defined types.
- Functions may be declared multiple times with consistent types. Functions that are referenced (and not library functions) must be defined exactly once. Structs may be declared multiple times, but may be defined at most once. Structs declared in libraries cannot be defined. Type names may be defined only once (they cannot be declared).
- A function **int main()**; is implicitly declared and also implicitly referenced, because this is the function called when an executable resulting from compilation is invoked. Therefore, when a collection of sources is compiled, at least one of them must define `main` to match the above prototype.

The return value of `main` is not treated as an indication of success or failure of the program as it is in C; any program that returns from `main` prints the returned integer and signals to the operating system that the program completed successfully, whereas any program that returns by executing the **error** statement signals to the operating system that it completed unsuccessfully.

- Field names within each struct must be pairwise distinct.
- Expressions `*NULL` are disallowed.
- Type **void** is used only as the return type of functions.
- Expressions, used as statements, must have a small type or **void**.
- Undefined structs cannot be allocated.
- **continue** and **break** statements can only be used inside loops.
- The step statement in a for loop may not be a declaration.
- Integer constants are in the range from 0 to 2^{31} .
- `* <lv> ++` and `* <lv> --` must be explicitly parenthesized to override the right-to-left associative interpretation of `++` and `--`.

In addition we check in annotations:

- `\result` is only legal in `@ensures` clauses.
- `@requires` and `@ensures` can only annotate functions.
- `@loop_invariant` can only precede loop bodies.
- `@assert` can not annotate functions
- Expressions occurring in function annotations can only refer to the functions parameters. Expressions in loop invariants and assertions can also use other local variables in whose scope they occur. Variables in `@ensures` clauses cannot be assigned to in the body of the function they annotate.

17 Updates

Jan 30 2011. A stand-alone semicolon `;` is now flagged as an error rather than interpreted as an empty statement. Remember that conditionals, loops, and blocks are *not* terminated by a semicolon. Use an empty block `{}` as a statement with no effect.

Dec 8 2012. Added the error statement to the language and clarified that returning a non-zero integer from `main` does not signal unsuccessful execution to the operating system, though running the error statement does.

Dec 18 2012. Left and right shift operations now require their second operand k to be in the range $0 \leq k < 32$. Otherwise, an arithmetic exception is raised.

Dec 22 2012. `#use` directives must precede all other declarations.

Dec 22 2012. Statement blocks may end in annotations.

Dec 27 2012. Conditional expressions must have small type.

Dec 27 2012. Documented purity checking.

Sep 1 2013. Moved **`break`** and **`continue`** from C0 to an extended language standard C1. They remain reserved keywords in C0.

Aug 21 2014. Removed support for `\old(e)` in contracts, specifically `@ensures` clauses.

Aug 21 2014. Added generic pointers (**`void*`**) and function pointers to C1 to permit generic implementations of data structures.

Rev. C0.0590, March 30, 2018. Reformatted documentation to use the `listings` Latex package.

Rev. C0.0720, August 20, 2020. Modified the lvalue rules to allow casts from **`void*`**